Software Tech News

Volume 2 Number 1

STN 2-1 Topic:
Rapid Application Development (RAD)

In This Issue **:**

Rapid Application Development (RAD): A Brief Overview

by Morton A. Hirschberg - U.S. Army Research Laboratory

Introduction

Rapid Application Development (RAD), a revolutionary software archetype of the 1990's, while living up to its promise is still a fertile area for continued research and additional capitalization. This is evidenced by recent workshops at the University of Southern California; Center for Software Engineering (June 1997 and March 1998), the Software Productivity Consortium Workshop in Herndon, VA (November 1997), the Software and Systems Engineering Productivity Project of the Microelectronics and Computer Technology Corporation, Austin, TX, and the Software Technology Conference Panel (April 1998). With little variation the tenets of RAD are essentially those of software development in general: methodology or choice of architectures and tools, requirements and design analyses, selection of personnel and management, construction, and implementation and support. What then sets RAD apart is a very structured approach typically relying on small well-trained teams, use of evolutionary prototypes, and rigid limits on development time fames. In summary, the goals of RAD are: faster, better, cheaper.

Continued on page 7

Your Source for Information in Software Engineering Technology.

DoD Data & Analysis Center for Software
http://www.dacs.dtic.mil

Rapid Prototyping: DACS Track at STC '98

DoD DACS Staff

The Tenth Annual Software Technology Conference (STC) will be held 19-23 April 1998 at the Salt Palace Convention Center in Salt Lake City, Utah. "Knowledge Sharing - Global Information Networks" is the theme of this year's conference. The DACS is sponsoring a session on Rapid Prototyping/ Rapid Application Development (RAD). RAD is also the theme of this issue of Software Tech News. Three of the four DACS guests to present papers at the STC summarize their positions in this issue. In addition, we present Morton Hirschberg's overview of RAD.

Mr. Larry Bernstein, founder of Have Laptop - Will Travel, argues there is no best way to produce software. He states that current theory is not adequate for analyzing the dynamic behavior of software systems under varying loads. He concludes by listing seven reasons that prototyping addresses fundamental needs of software processes.

Dr. Stephen E. Cross, director of the Software Engineering Institute (SEI), discusses progress in software engineering, risks associated with current practices, especially in the development of "unprecedented systems," and a RAD approach that addresses these risks. Dr. Cross emphasizes the need for a disciplined RAD methodology, and outlines a six step process to provide the needed rigor.

Major David A. Dampier, Ph.D., Professor at the National Defense University, Fort Lesley J. McNair, Washington, DC, argues for the need to proceed from prototypes to a system delivered to the user more quickly than is possible with today's technology. Major Dampier envisions the use of software tools to quickly produce prototypes that can be delivered to the user. These tools will be embedded in an incremental software development methodology.

Mr. Bernstein, Dr. Cross, and Major Dampier will elaborate their views further at the STC. In addition, Dr. Erik G. Mettala, Executive Vice President of MCC, will present his perspective on RAD at the STC.

STC Contact Information:

Ms. Dana Dovenbarger
dovenbar@oodis01.hill.af.mil

or

Ms. Lynne Wade
wadel@software.hill.af.mil

Voice: (801) 777-7411
DSN: 777-7411
Fax: (801) 775-4932

http://www.stc98.org

Visit the DACS Home Page for great resources on Rapid Prototyping and 16 other Software Technology Topic Areas.

http://www.dacs.dtic.mil

Don't forget to visit the DACS **booth (#238)** at STC '98!

Importance of Software Prototyping

Larry Bernstein - Have Laptop - Will Travel

Modern software development demands the use of Rapid Application Prototyping. Professor Luqi at the Naval Postgraduate School coined the term “Computer Aided Prototyping” to describe this work. Her leadership showed its effectiveness in gaining understanding of the requirements, reducing the complexity of the problem and providing an early validation of the system design. For every dollar invested in prototyping, one can expect a $1.40 return within the life cycle of the system development.

Dr. Barry Boehm's experiments showed that prototyping reduces program size and programmer effort by 40%. It is the technology that is the foundation for his Spiral development method. Prototyping is being used successfully to gain an early understanding of system requirements, to simplify software designs, to evaluate user interfaces and to test complex algorithms. It is a best-in-class software approach.

Fully 30 to 40% of system requirements will change without prototyping. Rapid Application Prototyping provides a look at the dynamic states of the system before we build it, whereas most other software engineering focuses on the source code. The special problems of reliability, throughput and response time as well as system features are addressed in the best prototypes. A new field of study, *Software Dynamics*, will emerge once Rapid Application Prototyping is widely practiced. It will focus on quantitative analysis of how software performs under various loads and include a set of design constraints that will make it possible for us to build components that can be hooked together without exhaustive coverage testing.

Software is hard because it has a weak theoretical foundation. Most of the theory that does exist focuses on the static behavior of the software, analysis of the source listing. There is little theory on its dynamic behavior, how it performs under load. To avoid serious network problems software systems are over-engineered with plenty of bandwidth for two or three times the expected load. Without analysis of the dynamic behavior, application designers have no idea of the resources they will need once their software is operational. Software has the awful propensity to fail with no warning. Even after we find and fix a bug, how do we restore the software to a known state, one where we have tested its operation? For most systems, this is impossible except with lots of custom design that is itself error-prone. Software prototyping has proven its mettle in helping designers avoid these problems in their production systems.

Much has been written about the best way to develop software applications. But there is no "best way." Both prototyping and requirements are necessary. The tried-and-true process of synthesis and analysis is used to solve software-engineering problems. Bottom-up is synthesis. Top-down is analysis. Bottom-up is prototyping. Top-down is developing requirements. Prototyping is the best way to encourage synthesis. Prototyping also eases communication with the customer and with the designer. Formal written requirements are needed to establish a clear definition of the job, to control changes and to communicate the system capabilities between the customer and the developer.

So where does this leave us? Start with an English language written statement of a problem and broadly outline its solution. Now build a prototype for the elements where you need insight. Analyze the prototype using computer aided prototyping

Continued on page 9

Rapid Prototyping and Incremental Evolution

David A. Dampier - National Defense University

Software development is no longer an enterprise where the traditional waterfall method of system construction is acceptable. Information technology is changing at a pace that requires complete system development and fielding in less than 18 months. This is due in part to faster technology insertion, and in part by increased user expectations. Both reasons provide justification for changing the way software is built and fielded. Increased user expectations require that we involve the user more in the requirements engineering process, and deliver the software to the user much more quickly. Faster technology insertion requires that we incorporate new technology into existing products much faster and with less rework.

A new software evolution paradigm is needed to accomplish these goals, along with the automated tools to realize the benefits. Computer-Aided Prototyping is one such method that incorporates the goals and opinions of the user from the beginning of the software evolution process, throughout the lifecycle, and into retirement. Automated tools, like the Computer-Aided Prototyping System (CAPS)[1], assist the software developer in building executable prototypes of a software system very quickly, involving the user in an iterative build-execute-modify loop until the user is satisfied with the demonstration of the prototype. The prototype is then used to build the final version of the software through the use of the architecture included in the prototype, as well as the validated set of requirements constructed during the prototyping process. The resulting final version is delivered relatively quickly, hopefully before the user's requirements have an opportunity to change.

This is often not sufficient to satisfy some users. If the demonstration of a prototype to a customer results in the validation of the requirements for that system, the user may want to take the prototype as is. Since most prototypes are not industrial strength, this may not be possible. The need outlined here is for a system, like CAPS, that will result in a version of the system that can be delivered to the customer immediately upon validation. The system could be used as it is until it no longer satisfies the user's requirements. When the user's requirements do change, new requirements can be incorporated into a next version of the system by using the same iterative process where the fielded version of the system provides the base version of the process. This incremental evolution process can proceed throughout the life of the system.

Professors Luqi and Berzins of the Naval Postgraduate School , along with a myriad of graduate students and visiting researchers, have spent many years developing CAPS [1]. Their efforts have resulted in a system that can be used to build executable prototypes of embedded real-time systems. These prototypes are useful for validating requirements through demonstrations to customers, but are not practical for providing the kind of deliverable version of a software system discussed in this article.

CAPS relies on external support for building graphical user interfaces, and manual translation of requirements into prototypes. This manual translation is problematic. The possibility of misinterpretation by the designer could lead to wasted effort in the prototype building process. Prototypes generated using CAPS generally lack robustness and portability. It is easy to build prototypes that do precisely as expected as long as the proper inputs are made, and use follows the designer's expectations. If improper inputs are made, and the designer has not built sufficient error handling into the

Footnotes:

1. Luqi and Ketabchi, M., "A Computer-Aided Prototyping System" *IEEE Software*, March 1988.

Continued on page 12

Toward Disciplined Rapid Application Development

Stephen E. Cross - Director, Software Engineering Institute

As the director of the Software Engineering Institute, I am often asked to explain why progress in software engineering has not kept pace with progress in other engineering fields. The advances in computer hardware, characterized by phenomenal increases in processor speed and memory capacity associated with decreases in size and cost are the most cited progresses. I contend that similar progress has been made in the engineering of software-intensive systems. I suggest an approach to disciplined Rapid Application Development (RAD) that builds on the progress made in software engineering during the past 30 years.

Consider briefly the progress of the past 30 years. During the 1970s, the age of "programming productivity," the creation of new high-order languages, tools, and development methodologies enabled programmers to improve their productivity by one to two orders of magnitude. During the 1980s, the age of "software quality," the focus was on software processes and continuous process improvement. Quality results have been published in the literature for the past couple of years (for example, see [1], [2], or [3]) and indicate improvements in an order of magnitude range along several dimensions (decreased defects, increased productivity, decreased cycle time, decreased number of personnel required to achieve results, and decreased percent rework after release). The decade of the 1990s is the age of "Internet time." The advent of the Internet and associated new software technologies (for example, Java and the widespread use of object technology) enables software developers to field products in cycle times of 6 months or less. The combination of best practices that have evolved over the past 30 years in productivity approaches, quality improvement, and technology is impressive and matches progress in other fields of engineering. Taken collectively, they form an arsenal of tools (rather than the proverbial "silver bullet") with which to attack software development.

While the progress is real and arguably impressive, the reasons for failures in software development are largely the same today as they were 30 years ago. In a 1988 U.S. Air Force Science Advisory Board Study [4], three common reasons were cited for failure (where failure ranges from excessive cost and/or schedule delays to never fielding a system).

1. **Risks associated with teams.** By this was meant that if a team of developers, acquirers, end users, and systems maintainers (and their management) had not worked together before and did not learn to communicate effectively, they were not likely to develop a successful system without schedule delays or cost overruns. Other risks cited were the lack of well-defined or well-understood processes.
2. **Risks associated with technology.** Teams that pursued a new technical approach (for example, the first foray into client-server computing) found that the lack of experience with a new technology, architecture, or development approach contributed to failure.
3. **Risk associated with requirements.** By far the most often-cited reason for failure was poor management of requirements characterized by frequently changing requirements, requirements that were not well understood, and requirements proliferation.

Continued on page 10

Book Review: Software Engineering: A Practitioner's Approach, Fourth Edition by Roger S. Pressman

Reviewed by Marshall Potter, Office of the Director, Defense Research and Engineering

This book, which is now in its fourth edition, is one of the classic texts on Software Engineering. It is designed for both the novice and experienced software engineer, and all readers will find an abundant amount of material from which they can learn. In this book, you will find not only cogent discussions on the key components of Software Engineering, but links and references for further study. One of its new innovative features is the integration of the World Wide Web addresses into the text. This makes the text a virtual encyclopedia. The fourth edition is considerably more than a simple update and provides a significant improvement by providing extensive coverage of both new and evolving strategies and technologies that are core to effective system developments.

The book is divided into 30 chapters that are organized into five parts. The first part covers both the software product and its process. Pressman carefully explains what software is and why we continue to struggle to build high quality systems. He goes into some detail on what myths continue to exist and categorizes them from three perspectives: management, customer and practitioner. This division is vitally important, as one can't develop high quality software without addressing all three perspectives. In chapter 2, he introduces both the SEI Software Process Model and the various process or life cycle models that have been used over the past three decades. These include a gamut of models including the linear sequential model (waterfall), the prototyping model, the Rapid Application Development (RAD) model, and several evolutionary process models including the incremental, spiral, component assembly, and concurrent development. He closes this section with an introduction on the formal methods model, fourth generation techniques and process technology. As can be seen, he covers not only what has been used, but also new and emerging techniques.

McGraw-Hill ISBN 0-07-052182-4

Part two of the text is devoted to Managing Software Projects. I especially welcome this introduction to management in the early part of the text. I believe that management and management science, from both the engineer's and computer scientist's perspectives, is often ignored or is added as an afterthought. Pressman by locating this important material up front, signals the importance he gives to the topic. The author addresses management from the perspective of the three P's; people, problem and process. In chapter 3, he exhibits how new efforts such as the SEI's People Management - Capability Maturity Model (PM-CMM), tools and techniques for addressing the problems and their risks and frameworks for process definition and execution are becoming common concepts in the most successful software development organizations. He addresses software management as the "umbrella activity" that begins before any technical activity is initiated and continues throughout the definition, development and maintenance of computer software. As the author truly knows, management is fundamental for success, and I would therefore have liked to have seen more emphasis and references on traditional management science. Other authors, such as Thayer and Reifer, have shown how the classical management model of planning, organizing, staffing, directing and controlling directly relate to successful software/ systems engineering programs. I would have liked to have seen this framework used in this section as it would have provided a unifying architecture for the material that Pressman discusses in the following chapters.

Continued on page 13

RAD: A Brief Overview

Continued from page 1

The Search for the Universal Architecture

Much as the search for the Holy Grail, software developers are continually seeking the universal architecture. One of the strategies of RAD is an up front investment in producing a suitable architecture and populating it with just the right tool suite. At the core of development is utilization, and although not a strict dictum, the use of the Spiral Model allows incremental and repetitive development. The use of Object-Oriented methods is encouraged to speed development and allow for reduced rework and possible reuse. Automated code generators such as the Computer-Aided Prototyping System (CAPS) replaces slow hand written code and minimizes coding errors. RAD also allows for users to employ their own query and update languages, report generators, decision support languages, as well as specification languages. This tailoring and flexibility in the rapid production of prototypes, products, and systems is mitigated by domain specificity which brings order out of apparent chaos.

Will my Ship Sail on Land as Well as the Sea

One of my favorite fairy tales revolves around manufacturing a ship that will sail on land as well as the sea. The successful inventor clearly states the requirements in the problem statement or statement of work (SOW) and then focuses on the design and then construction of such a vehicle. He stays on the critical path throughout the development without any sacrifice of high quality. In the end, the user's needs are fully met. Formal requirements establish a clear definition of the tasks. They also are used to communicate the system capabilities among the customer, user, and developer. Requirements should include design features, performance goals, and schedule and cost estimates. The use of the Software Engineering Institute's (SEI) Capability Maturity Model (CMM) can be an invaluable resource in suggesting what should be done by having a well defined and well understood processes. An important goal of RAD is to keep the time between design and delivery as short as possible. So the use of cost estimators such as the COCOMO model, to name but one, and PERT charts to stay on the critical path, to name another, are highly encouraged.

Who's on First

RAD depends upon continuous, high quality, production. The optimal is a team of users, acquirers and developers who can communicate effectively and successfully develop their products without schedule delays or cost over runs. To this end, as Dr. Cross points out, experience counts. Similarly, it is management's function to eliminate unnecessary tasks, streamline activities, and increase work time while the staff reduces time per task, and reduces or eliminates backtracking. Having a well trained, fully collaborative team is an essential ingredient for success. The core of the team should be full participants in project planning. The core of the team should stay together from start to finish. Support tools should be provided to those skilled in using them. Quality and configuration management should be imposed from within, anytime, anywhere development and the use of virtual offices provides an atmosphere for employee satisfaction.

Construction

This is the phase where prototypes are formed, products developed, and systems produced. It is the crucible of the architecture and tools and the staff and managers who mold and construct them. This is what we have been waiting for, the answers to our questions. We can see the effects of inputs on outputs and marvel at sometimes unexpected but correct results. We also see faults and shortcomings as well. We can determine how robust our products and systems are and if prototypes should be further

Continued on page 8

RAD: A Brief Overview

Continued from page 7

developed. We can assess risk with far greater accuracy and project the shelf life of our efforts. We can see the quality of our work. We can mark our progress towards meeting time to market, determine our status relative to our competitors, record the time to mature new processes and estimate if our efforts will scale if they are prototyped. We can save elements which can be reused. We can consider how new systems can improve our business and streamline our processes and procedures. We can see if we are truly generating new and valuable information.

It's Not Over Until It's Over

Once we have crossed the construction hurdle and decided to continue we enter the implementation and support phases. In other words, coding, use and maintenance. The latter, as we know, can be 90% of the entire life cycle cost. It is here where we continue our metrics collection but with the counsel of our users. It is here where we continue to respond to requirements and design changes, make modifications, corrections, and improvements. It is here where we assess our true costs and profits (hopefully no losses) and calculate our Return On Investment (ROI). It is here where we begin to plan for the future.

Conclusions

RAD has proven to be a valuable software strategy. It is not without pitfalls and risks. It requires the right mix of methodologies, tools, personnel and management. Its use depends upon complexity of the domain or application, the organizational environment, the skills of staff and management and the architectures and infrastructures available. RAD Is worthy of continued research and capitalization.

About the Author

Mr. Hirschberg has over 40 years experience in Software Engineering, 25 years in the government sector. He has authored over 50 papers.

Morton A. Hirschberg
U.S. Army Research Laboratory
Aberdeen Proving Ground,
Maryland 21005

mort@arl.mil

http://www.arl.mil/

References

Boehm, Barry, Devnani-Chulani, Sunita and Egyed, Alexander, Editors; "Knowledge Summary: Focused Workshop on Rapid Application Development", USC, Los Angeles, 23-27 June 1997.

The Software Productivity Consortium. *The 5th Member Forum Proceedings: Technologies for the Rapid Development of Software*, SPC-97091-CMC, Herndon. VA, November 1997.

The University of California, Davis; "Application Development Methodology";
http://irlinux.ucdavis.edu/BillA/WEBADM/index.htm

Importance of Software Prototyping

Continued from page 3

technology and synthesize a new solution either by refining the prototype or building a new one. Once you and the customer agree on the workings of the prototype, write requirements that include features, performance goals, product costs, product quality, development costs and schedule estimates.

What do I mean by prototype? Prototyping is the use of approximately 30% of the ultimate staff to build one or two working versions of various aspects of a system. It is not production code but it may eventually become pre-production code or it may be completely discarded. In the prototyping effort, we are not concerned with the maintainability of the code nor are we concerned with formally documenting it. Code resulting from prototyping is often used to train the programmers. Only after we have written specifications resulting from the experience with the prototype should we start the formal development process. If we are fortunate enough that some of the code that was developed for the prototypes can be carried forward, that's great, if not, there is no loss.

A prototype produces "running" software and the production development produces "working" software.

Recent project experience has led to the widespread acceptance of the concept that early prototyping is fundamental to the success of operations supporting software products.

The reasons why prototyping is fundamental include:

1) The prototype provides a vehicle for systems engineers to better understand the environment and the requirements problem being addressed.
2) A prototype is a demonstration of what's actually feasible with existing technology, and where the technical weak spots still exist.
3) A prototype is an efficient mechanism for the transfer of design intent from system engineer to the developer.
4) A prototype lets the developer meet earlier schedules for the production version.
5) A prototype allows for early customer interaction.
6) A prototype demonstrates to the customers what is functionally feasible and stretches their imagination, leading to more creative inputs and a more forward-looking system.
7) The prototype provides an analysis test bed and a vehicle to validate and evolve system requirements.

About the Author

Mr. Bernstein is president of the Center of National Software Studies and is a recognized expert in Software Technology. He provides consulting through his firm Have Laptop - Will Travel and is the Executive Technologist with Network Programs, Inc. building software systems for managing telephone services.

Mr. Bernstein was an Executive Director of AT&T Bell Laboratories where he worked for 35 years.

lberstein@worldnet.att.net

Toward Disciplined Rapid Application Development

Continued from page 5

The bottom line is that experience counts. The study coined the term "unprecedented systems" to describe systems in which these risks were present. An experienced team, developing a similar system to one that it has previously developed, with a customer and end user with whom it can communicate well, is much more likely to produce high-quality software-intensive systems on time and at cost.

With this as backdrop, I contrast my own experience as a computer scientist and software engineer. My formal training (some would say "formal" is too strong given that my graduate work was in machine intelligence) focused on Rapid Prototyping. This was during the late 1970s and early 1980s, the early and exciting days of the first commercial expert systems. In the laboratory, our research prototypes were useful tools for experimental research. To our commercial counterparts, rapidly developed prototypes were often "throwaways." They were often too fragile to scale into a hardened, deliverable system. But they served a critical purpose, they enabled one to quickly capture an explicit and inspectable representation of requirements and depict them in a meaningful way to end users. The tools of the day allowed one to work interactively with end users to evolve a more complete understanding of those requirements. In effect, they provided a means of communication through which a development team (including users, maintainers, and management) could discuss and reach common understanding of the requirements.

Many have criticized Rapid Prototyping, or as it is now more frequently called, Rapid Application Development (RAD)—as lacking rigor, leading to fragile systems that do not scale, and serving to raise end user and management expectations to unrealistic levels. These criticisms are valid, unless a more disciplined approach to RAD is followed that couples RAD with the lessons learned in productivity and quality. The approach I propose is based on Boehm's spiral model [5]. In the spiral model, a complete representation of the system is produced and tested during each development cycle (or spiral). Each spiral addresses a particular risk, with the most serious risks addressed in the earliest cycles. I have used this approach in several successful systems [6,7], and variations have also been discussed in the literature [8].

A proposed approach to disciplined RAD would entail these steps:

1. Scenario-based design and analysis
2. Architecture design and analysis
3. Component specification with maximum reuse
4. Rapid development of remaining modules
5. Frequent testing with end users and systems personnel
6. Field with support tools to allow for evolution

The progress in software technology now makes this approach much more likely. Step 1 addresses the major source of risk described, requirements. Scenario-building tools allow rapid development of cases to illustrate system operation, which in turn are useful for defining, refining, and communicating an understanding of requirements. Because end users and management often see ways to improve their work processes as a result, this approach has also proven useful in business reengineering. A by-product of this approach is the capture of test cases that can be used for user-centered testing at later stages in the system development. Thus, scenario-based approaches provide a useful way to do requirements analysis.

Steps 2 and 3 address technology risks. As in other engineering fields, it is useful to define the architecture early during system development and to conduct

Continued on page 11

Toward Disciplined Rapid Application Development

Continued from page 10

trade-off analysis to assess attributes such as data throughput, usability, and security issues. Too many past failures are be attributed to failure to understand a technical constraint until realization of the software system in executable code. Recent advances in software architecture development and analysis (for example see [9]) provide an engineering basis for early architecture specification. In addition, a lesson learned from reusable software development is the criticality of software architecture in which to embed reusable software components. Components that do not exist or that cannot be easily retrofitted into the architecture can be developed using a rapid prototyping approach (step 4). Requirements and architecture provide design constraints to bound and guide the development of these modules.

Steps 5 and 6 are also very important. It is critical that end users and system maintainers participate regularly in testing. Though I list it as a separate step (a final test before delivery needs to be done) it is also useful to use scenario-based test data to assess the output of each step. Lastly, as requirements will change over the life cycle of the system, it is important to consider how systems will be used and will likely evolve, and then plan for that evolution.

Not explicit in the above approach is mitigation of risks associated with people and process. It is my belief that process improvement, under such approaches as the Capability Maturity Model (sm) for Software [10], is not inconsistent with RAD. The Capability Maturity Model (CMM) suggests what should be done, not how to do it. The discipline in the above approach comes from having well-defined and understood processes. In addition, training for new employees and continuing education for all employees is an important aspect to ensure that the development team can cope with technical change.

So how will we characterize the first decade of the new millennium? Trends suggest we will have more powerful computing coupled with a low-cost, high-bandwidth communication infrastructure. There will be continued downsizing of organizations and more outsourcing. There likely will be marketplaces for reusable objects and software components. My bet is that a disciplined RAD approach will become the de facto approach for the development of software-intensive systems.

About the Author

Stephen E. Cross - Director,
Software Engineering Institute

Carnegie Mellon University
Pittsburgh, PA 15213-3890
sc@sei.cmu.edu

http://www.sei.cmu.edu/

The Software Engineering Institute

Toward Disciplined Rapid Application Development

Continued from page 11

References

1. Diaz, M and Sligo, J. "How Software Process Improvement helped Motorola," *IEEE Software*, September 1997, vol.14, no.5, p. 75-81.
2. Haley, T. "Raytheon's Experience in Software Process Improvement," *IEEE Software*, November 1996, Vol. 13, No. 6, November p. 33-41.
3. Fox, C. and Frakes, W. (Eds), Special Issue on the Quality Approach: Is it Delivering?, *Communications of the ACM*, June 1997, vol. 40, no. 6.
4. Sylvester, R.J. and Stewart, M. "Unprecedented Systems," *Encyclopedia of Software Engineering*, Marciniak, J. (Ed), J. Wiley, 1994.
5. Boehm, B. "A Spiral Model of Software Development and Enhancement," *Computer*, May 1988, vol. 21, no. 5, p. 61-72.
6. Wiederhold, G. and Cross, S. "Alternatives for Constructing Computing Systems," in *Computers as Our Better Partners*, Yahiko Kambayashi (Ed), ACM Japan Symposium, World Scientific Book Co., pp. 14-21, 1994.
7. Cross, S. and Estrada, R. "DART: an Example of Accelerated Evolutionary Development," *Proceedings of the Fifth IEEE International Workshop on Rapid System Prototyping*, Villard de Lans, France, June 1994.
8. DeBellis, M. and Haapala, C. "User-Centric Software Engineering," *IEEE Expert*, February 1995, vol. 10, no. 1.
9. Bass, L., Clements, P., and Kazman, R. *Software Architecture in Practice*, Addison Wesley, 1998.
10. See http://www.sei.cmu/technology/

sm - Capability Maturity Model is a service mark of Carnegie Mellon University.

Rapid Prototyping and Incremental Evolution

Continued from page 4

prototype, it is possible that execution can halt unexpectedly. Robustness can be built into CAPS prototypes, but automated methods for testing these qualities are not included. Manual methods are possible, but would severely increase development time.

Portability is a different matter. Current versions of CAPS run on SunOS and Solaris, but most software in use by the military runs on PCs. This means that prototypes built using CAPS could be demonstrated to the customer on UNIX machines, but would have to be translated into something that would run on PCs. Although this is not a severe limitation, it does make it difficult to deliver the prototype to the customer immediately upon validation.

Current software development methods and tools are insufficient to produce usable code in a reasonable amount of time. Rapid prototyping methods approach the needed capability, but are not yet up to the task. An incremental software development methodology, modelled after the rapid prototyping paradigm used in a system like CAPS, is needed to reduce development time and put something usable in the hands of the customer quickly.

About the Author

Major David A. Dampier is a professor at the National Defense University in the Information Resources Management College.

He may be contacted at:

Major David A. Dampier Ph.D.
National Defense University
Building 62, 300 5th Avenue
Fort Lesley J. McNair,
Washington, D.C. 20319

http://wwwndu.edu/

Book Review of Software Engineering: A Practitioner's Approach

Continued from page 6

In chapter 4 of the text, we find a good introduction to the software process and project metrics. It is gratifying to see his early introduction to Humphrey's Personal Software Process (PSP) prior to addressing the project oriented metrics of size and quality. Pressman provides more emphasis on function-oriented size metrics than on the traditional line-of-code models, but provides ample links to these at the end of the chapter. Additional material on the more technical/product oriented metrics is found later in the text.

In chapter 5, the author covers software project planning. This shows the intimate relationship between the material introduced in chapter 4 on metrics and measurement to the planning process. Pressman's primary emphasis is on estimating, and he covers both line of code and function point based models. He emphasizes five questions that need to be answered: (1) How long will it take? (2) How much effort is required? (3) How many people will be involved? (4) How many resources (Hardware/ Software) will be required? (5) What are the risks and how do we manage the known risks? Moving on from planning and estimating, Pressman provides good coverage of the current thinking on Risk Management in chapter 6. In this chapter you are introduced to Risk Management concepts used by Pressman, Charette, Boehm, the SEI, the Air Force and others. Unlike previous texts, there is considerable attention given to this important topic and this adds much to the value of the book. Chapter 7 is devoted to Project Scheduling and Tracking which obviously draws much upon the material covered in the previous chapters. Chapter 8 on Software Quality Assurance addresses SQA from the viewpoint that quality assurance encompasses (1) a quality management approach, (2) effective software engineering technology, (3) formal reviews that are applied throughout the process (4) a multitiered testing strategy, (5) control of software documentation and the changes made to it, (6) a procedure to assure standards compliance and (7) measurement and reporting mechanisms. As can be seen from this list, Pressman continues to build upon what he has introduced before. By doing this, Pressman provides the reader with a cohesive and structured viewpoint of software engineering based upon a core set of principles that are elaborated upon as you traverse the text. However, there is an unfortunate omission in this section with the absence of a mention of Michael Fagan and the Fagan Inspection Process. I feel that a student using this text without a teacher, would never come upon the name of this important figure or the inspection process which is often named after him. Many of the references in the back of this chapter refer to Fagan's work, yet not one reference is directed to Fagan's original papers.

Chapter 9 concludes this section by providing a well structured overview of configuration management.

Part Three of the text is devoted to Conventional Methods for Software Engineering concentrating on Systems Engineering, Analysis, Design, Testing and Metrics. As can be expected, this large area demands the most coverage and over 300 of the book's 800+ pages is devoted to this core area. When you consider that Part Four is devoted to Object Oriented Techniques and takes an additional 120+ pages, over half of the text is devoted to the core concepts of analysis, design and testing. This is very appropriate and provides good coverage to current methodologies that the software engineering practitioner needs to know. Relatively new techniques in Quality Function Deployment (QFD), modeling and prototyping are well covered in chapter 11. Chapter 12 introduces various methodologies and their notations so that the reader is introduced to a variety of analysis modeling techniques. Chapter 13 introduces the design concepts of abstraction, refinement, modularity, architecture, hierarchy, structural partitioning, data structure, software procedures and information hiding. From there, effective modular design based on functional independence measured by cohesion and coupling concepts is well

Continued on page 14

Book Review of Software Engineering: A Practitioner's Approach

Continued from page 13

covered. Chapter 14 provides an extensive coverage of various design methods and chapter 15 extends these for real-time systems. Chapters 16 and 17 are devoted to testing techniques. Chapter 18 expands on the material in Part Two that was devoted to measurement and metrics and focuses on technical metrics in contrast to management metrics.

Part Four of the text, as noted above, proceeds from Part Three and tackles the important area of Object-Oriented Software Engineering. Pressman covers analysis, design and testing of object-oriented software, but does not discuss object-oriented programming. As this subject will be covered in most Computer Science curriculums in the programming courses, this seems to be a prudent choice and helps keep the size of the text down. Coverage of a variety of methods including Booch, Coad and Yourdon, Jacobson, Rumbaugh and Wirfs-Brock methods are outlined in chapter 20. The important area of object-oriented testing is covered in chapter 22.

Part Five of the text discusses several advanced topics with separate chapters devoted to formal methods, cleanroom, software reuse, reengineering, client-server software engineering and Computer Aided Software Engineering or CASE. The book concludes with an exploration of the scope of change in the field and how this change itself will affect the software process in the future.

This impressive work provides an virtual encyclopedia of the present state of software engineering with several unique innovations. First is the use of the World Wide Web for references. Second is the fact that Pressman keeps these references up-to-date on his own web site at http://www.rspa.com. And finally, this text attempts to look at Software Engineering from a holistic and structured perspective, providing a travel guide or roadmap that takes you from one place and builds upon the concepts developed as you move from chapter to chapter. Because of this, I believe that the book is best read, at least the first time, cover to cover. There are however some minor problems and obstacles that should have been found in the initial proof reading. Several of the URLs noted in the back of each chapter and on his web site are incorrect. As an example, in chapter 2, the URL for the Software Productivity Consortium is shown to be http://software.software.org/vcoe/home.html and it should be http://www.software.org/vcoe/home.html. Several other typos were found. On p.585 James Rumbaugh is spelled Rambaugh and the negative impact of this typo is compounded by the fact that the reference on p. 611 is also for Rambaugh. Mistakes like these make it very difficult for the student to use the references. I need to note, that I was using the first print for my review. These mistakes are being corrected in a soon to be issued third printing and errata sheets for the first two additions are available at Pressman's Web site, http://www.rspa.com. I strongly recommend that you get the latest printing. Even considering these errors, the text takes a giant step forward and provides the model for all future software engineering texts. It is a virtual encyclopedia that can be used to teach both up-coming and experienced software engineers. I have used Pressman's previous editions in my classes on software engineering. I find this edition to be a major improvement. With the errors being corrected, this book will become an instant classic. This is a truly exceptional reference work that deserves to be on every software engineer's shelf and used regularly. It should be noted that a complete video curriculum is available to be used with the text. These videos would be helpful if you are planning to cover this material for large organizations.

About the Reviewer

Marshall Potter is an Adjunct Associate Professor at the University of Maryland.

Marshall Potter
Special Assistant for Computing and Software Technologies
Office of the Director, Defense Research and Engineering, Information Technologies Directorate
pottermr@acq.osd.mil

The Dynamic Secure STINET Service Adds New Features

Pat Tillery - DTIC Product Management Branch

STINET now has added the following…

Secure STINET's Customization provides the power to create and modify your own personalized web page. See what has changed in STINET by filtering out what is old and concentrating on what is new…set up a personal profile based on subject fields and groups and automatically receive citations via E-Mail to the latest accessions in DTIC's Technical Report collection twice a month…save search queries for both the Technical Report and Work Unit Information System collections for reuse.

Abstracts are now included with citations to unclassified/limited documents in the Technical Reports Bibliographic Database. Viewing abstracts is based on individual user profile access restrictions. If your profile does not permit you to view a particular citation's abstract, you will be allowed to view the rest of the citation, minus the abstract.

Over 3,000 ***full-text technical reports*** are now available for viewing and downloading.

Special Collections highlights reports found in DTIC's Technical Reports collection based on the source, topic, or targeted group. In addition to setting up your own search parameters, you can search using preestablished profiles developed by retrieval experts.

The Partnership for Peace Information Management System (PIMS) is designed to enhance the education of U.S. Service school students. Topic searches developed by DTIC for the PIMS community provide information ranging from air traffic control management to public affairs. PIMS also offers students the capability to construct custom searches for information not covered in the topic searches.

Subscribe Today

The subscription for the Secure STINET Service access via a web client is $50 per year/per subscriber.

To subscribe to Secure STINET Service, contact DTIC's Registration Branch:

Telephone: (703) 767-8272, DSN 427-8272

Toll Free: (800) 225-3842 (menu selection 2, option 2, sub-option 2)

Fax: (703) 767-8228, DSN 427-8228

E-Mail: reghelp@dtic.mil

Direct Questions concerning this product to:

Product Management Branch, DTIC-BCP, (800) 225-3842 (menu selection 2, option 3), (703) 767-8267, or DSN 427-8267.

STINET is a Defense Technical Information Center (DTIC) Service !

DoD DACS Products & Services Order Form

Name: Position/Title:

Organization: Acronym:

Address:

City: State: Zip Code:

Country:

Telephone: Fax:

E-mail:

Product Description	*Format*	*Quantity*	*Price*	*Total*
The DACS Information Package				
❑ Including: Software Tech News newsletter, an introduction to the DACS and a Products & Services Catalog	Document		FREE	FREE
Empirical Data				
❑ Architecture Research Facility (ARF) Error Dataset	Disk		$ 50	
❑ NASA / Software Engineering Laboratory (SEL) Dataset	CD-ROM		$ 50	
❑ NASA / AMES Dataset	CD-ROM		$ 50	
❑ Software Reusability Dataset	Disk		$ 50	
❑ DACS Productivity Dataset	Disk		$ 50	
Technical Reports		*FREE with Spreadsheet* →		
❑ A Business Case for Software Process Improvement	Document		$ 25	
❑ ROI from Software Process Improvement Spreadsheet	Diskette		$ 40	
❑ A History of Software Measurement at Rome Laboratory	Document		$ 25	
❑ An Analysis of Two Formal Methods: VDM and Z	Document		$ 25	
❑ An Overview of Object-Oriented Design	Document		$ 25	
❑ Artificial Neural Networks Technology	Document		$ 25	
❑ A Review of Formal Methods	Document		$ 25	
❑ A Review of Non-Ada to Ada Conversion	Document		$ 25	
❑ A State of the Art Report: Software Design Methods	Document		$ 25	
❑ A State of the Art Review: Distributable Database Technology	Document		$ 25	
❑ Electronic Publishing on the World Wide Web: An Engineering Approach	Document	*SALE Item!* →	**$ 5**	
❑ Object Oriented Database Management Systems	Document		$ 25	
❑ Software Analysis and Testing Technologies	Document		$ 25	
❑ Software Design Methods	Document		$ 25	
❑ Software Prototyping and Requirements Engineering	Document		$ 25	
❑ Software Interoperability	Document		$ 25	
❑ Software Reusability	Document		$ 25	
Bibliographic Products				
❑ Rome Laboratory Research in Software Measurement	Document		$ 25	
❑ DACS Custom Bibliographic Search	Diskette		$ 40	
❑ DACS Software Engineering Bibliographic Database (SEBD)	CD-ROM		$ 50	

Method of Payment:

❑ Check ❑ Mastercard ❑ Visa

Number of Items Ordered ☐ *Total Cost* ☐

Credit Card # _____ Expiration Date _____

Name on Credit Card _____ Signature _____

Mail this form to: DACS Customer Liaison
Data & Analysis Center for Software
P.O. Box 1400, Rome, NY 13442-1400

Telephone: (315) 334-4905
Fax: (315) 334-4964
E-mail: **cust-liasn@dacs.dtic.mil**